# BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images, please do not report the images to the Image Problems Mailbox.**

```
void Request::service()
{
   const char *p = strchr(request, ' ');
   if( p )
      fileName = CString(request, p - request);
   else
      fileName = request;

   {
      const char *p = fileName;
      if( *p == '/' )
         p++;
      if( *p == 0 ) {
         // send default
         //sendFile("k:\\my documents\\internet address finder\\iafmain.htm");
#if defined(_IAF)
         sendFile("c:\\iaf\\html\\iafmain.htm");
         return;
#endif
      } else {
         if( strchr(p, '\\') == 0 && strstr(p, "..") == 0 ) {
            if( strchr(p, '/') != 0 ) {
               CString f = "c:\\ian\\";
               f += p;
               sendFile(f);
               return;
            }
            else
            {
#if defined(_IAF)
               CString f = "c:\\iaf\\html\\";
#elif defined(_MANAGE)
               CString f = "c:\\ian\\manage\\";
#else
               ASSERT(FALSE);
               CString f = "jakldf";
               //CString f = "k:\\my documents\\ad federation\\";
#endif
               f += p;
               sendFile(f);
               return;
            }
         }
      }
      sendError(c, "404 Not Found");
   }
}

void Request::sendInternalError()
{
   sendError(c, "500 Internal Server Error");
}
```

```
// rememberad.cpp
//

#include "stdafx.h"
#include "objects.h"
#include "rememberad.h"
#include "/d/toolkit/hashwf.h"
#include "/d/toolkit/crit.h"

const SZ = 107311;

// this is a test
static int cr;
#define INCRIT { ASSERT(cr++==0); cr++; }
#define OUTCRIT { ASSERT(cr--==1); cr--; }

void message(const char *);

extern CriticalSection fast;

struct Key
{
   DWORD userID;
   DWORD fromHash;

   BOOL operator==(const Key& k) const
   {
      return userID == k.userID && fromHash == k.fromHash;
   }

   void setID(User *u)
   {
      if( u->userID )
         userID = u->userID;
      else
         userID = u->ip;
   }

   void setFrom(const char *from)
   {
      fromHash = hashwf(from);
   }
};

UINT HashKey(Key key)
{
   return key.userID ^ key.fromHash;
   // default identity hash - works for most primitive values
// return ((UINT)(void*)(DWORD)key) >> 4;
}

struct Value
{
   DWORD adSent;
   DWORD time;
};

class Memory
{
public:
   Memory() : sent(100)
   {
      sent.InitHashTable(SZ);
   }

   void remember(Key& k, DWORD adID);
   DWORD lookup(Key& k);

private:
   void purge();

   CMap<Key, Key&, Value, Value&> sent;
} memory;

// rmin: fix
```





```
// todo: nonunique hashes
//
//DWORD hash(const char *from, User *u)
//{
// char buf[10];
// wsprintf( buf, "%lx", u->getID() );
// CString s = buf;
// s += from;
// return hashwf(s);
//}

void Memory::remember(Key& k, DWORD adID)
{
   static int count;
   if( ++count > 1000 ) {
      count = 0;
      purge();
   }

   Value v;
   v.adSent = adID;
   v.time = ::GetTickCount();
   sent.SetAt(k, v);
}

DWORD Memory::lookup(Key& k)
{
   Value value;
   if( sent.Lookup(k, value) ) {
      return value.adSent;
   }
   return 0;
}

void Memory::purge()
{
   const LIMIT = 1000 * 60 * 60 * 24;      // too much?

   if( sent.GetCount() > SZ ) {
      message("remember map > SZ");
   }

   DWORD now = ::GetTickCount();
   POSITION p = sent.GetStartPosition();
   while( p ) {
      Key k;
      Value v;
      sent.GetNextAssoc(p, k, v);
      if( now - v.time > LIMIT )
         sent.RemoveKey(k);
   }
}

void rememberSend(Ad *ad, User *u, const char *fromDoc)
{
   Crit c(fast);
// INCRIT
   Key k;
   k.setID(u);
   k.setFrom(fromDoc);
   memory.remember(k, ad->id);
// OUTCRIT
}

DWORD queryAdSent(User *u, const char *fromDoc)
{
   Crit c(fast);
// INCRIT
   Key k;
   k.setID(u);
   k.setFrom(fromDoc);
   DWORD d = memory.lookup(k);
// OUTCRIT
   return d;
}
```

```
   // a truly random distribution is used for them rather than
   // leftovers.
   static int testCounter;
   if( ++testCounter % 4 == 0 ) {      // just try every 4 to save CPU
      // test ad avail?
      lowestSI = 1051;
      int i = start;
      while( 1 ) {
         Ad& ad = *ads.GetAt(i);
         if( ad.type == Test && ad.si < lowestSI && ad.criteriaOK(db, user, page) )
         {
            lowestSI = ad.si;
            adLowestSI = &ad;
         }
         i = (i + 1) % nAds();
         if( i == start )
            break;
      }
      if( lowestSI <= 1050 )
         return adLowestSI;
   }

   lowestSI = SIMAX;
   adLowestSI = defaultAd;

   // Check remnants first.  This way, we don't
   // have to do ad matching for any targeted ads
   // with high SI's.
   //
   int i = start;
   while( 1 ) {
      Ad& ad = *ads.GetAt(i);
      if( ad.type == Normal && !ad.isTargeted() && ad.si < lowestSI && ad.spreadOK(page) )
      {
         lowestSI = ad.si;
         adLowestSI = &ad;
      }

      i = (i + 1) % nAds();
      if( i == start )
         break;
   }

   // this is temp; eventual all placements will have book rates
   // you'll want to remove this to get better performance (no ad matching
   // if remnant has worst SI).
   static int counter;
   if( ++counter & 1 ) {
      // for ads with no booking amount,
      // allow a targeted ad to run sometimes
      if( lowestSI == 1100 )
         lowestSI--;
   }

      // for ads where we don't care about # impressions,
      // bias in favor of targeted
//    if( lowestSI == 1100 )
//       lowestSI--;

   // todo later: if ads are sorted by si (lowest first),
   //             you can quit matching as soon as you find
   //             one.  Could be a good optimization.

   // do targeted
   i = start;
   while( 1 ) {
      Ad& ad = *ads.GetAt(i);
      if( ad.type == Normal && ad.isTargeted() &&
         ad.si < lowestSI &&
         ad.spreadOK(page) &&
         ad.matches(user, page) &&
         ad.exposuresOK(db, user) )
      {
         // found a good one
         lowestSI = ad.si;
```



```
         adLowestSI = &ad;
      }

      i = (i + 1) % nAds();
      if( i == start )
         break;
   }

   if( lowestSI > 1400 ) {
      // do either a barter ad or an fan dev ad
      static int counter;
      if( ++counter % 5 == 0 ) {
         // do an fan dev ad
         i = start;
         while( 1 ) {
            Ad& ad = *ads.GetAt(i);
            if( ad.type == FanDev && ad.criteriaOK(db, user, page) ) {
               // found a good one
               adLowestSI = &ad;
               break;
            }
            i = (i + 1) % nAds();
            if( i == start )
               break;
         }
      }
      else {
         // do barter
         lowestSI = SIMAX;
         i = start;
         while( 1 ) {
            Ad& ad = *ads.GetAt(i);
            if( ad.type == Barter &&
               ad.si < lowestSI &&
               ad.criteriaOK(db, user, page) ) {
               // found a good one
               adLowestSI = &ad;
               lowestSI = ad.si;
            }
            i = (i + 1) % nAds();
            if( i == start )
               break;
         }
      }
   }

   return adLowestSI;
}
```

```
// request.cpp
//

#include "stdafx.h"
#include "/d/toolkit/sock.h"
#include "request.h"
#include "/d/toolkit/iaf_util.h"

#if defined(_CONSOLE)
#include <fstream.h>
#endif

#if defined(_IAP)
extern ofstream *outLog;
void impression();
#endif

extern CString gratuitous;

Request::Request(
   Connection *_c,
   Verb _v,
   const char *_request,
   const sockaddr_in& from) :
      c(_c), request(_request), v(_v)
{
   userIP = from.sin_addr.s_addr;
}

int spider = 0;

BOOL Request::sendFile(const char *fileName, const char *insertStr)
{
#if defined(_IAP)
   *outLog << "-end " << fileName << " " << inet_ntoa( (in_addr&) userIP ) << '\n';
#endif

   const char insertChar = '-';
   BOOL isSpider = FALSE;

   CString hdr = "HTTP/1.0 200 OK\r\nContent-Type: ";
   if( strstr(fileName, ".class") != 0 ) {
      hdr += "application/java\r\nContent-Length: ";
   }
   else if( strstr(fileName, ".gif") != 0 ) {
      hdr += "image/gif\r\nContent-Length: ";
   } else {
      hdr += "text/html\r\nContent-Length: ";
#if defined(_IAP)
      impression();
#endif
      int gnt = 0;
      if( strstr(request, "-Agent: Lycos") != 0 )
         gnt = 1;
       if( strstr(request, "InfoSeek Robot") != 0 )
         gnt = 2;
      if( strstr(request, "-Agent: WebCrawl") != 0 )
         gnt = 3;

      if( gnt )
      {
         isSpider = TRUE;
         spider++;
         #if defined(_CONSOLE)
         cout << "********** Robot " << gnt << " **********\n";
         #endif
      }
   }

   const BUFSIZE = 130000;
   char buf[BUFSIZE + 2001;
   CFile f;
   CFileException fe;
```





```
   if( v == GET || v == POST ) {
      if( !f.Open(fileName, CFile::modeRead | CFile::shareDenyWrite, &fe) ) {
         if( fe.m_cause == CFileException::accessDenied )
            sendError(c, "404 Not Found (Access Denied)");
         else if( fe.m_cause == CFileException::sharingViolation )
            sendError(c, "404 Not Found (Sharing Violation)");
         else
            sendError(c, "404 Not Found");
         return FALSE;
      }
      n = f.Read(buf, BUFSIZE);
   }
   else {
      isSpider = FALSE;

      // HEAD
      n = getFileSize(fileName);
      if( n == 0 ) {
         sendError(c, "404 Not Found");
         return FALSE;
      }
   }
   ASSERT( n != 0 && n != BUFSIZE );

   char *p = buf;
   if( insertStr ) {
      while( 1 ) {
         p = strchr(p, insertChar);
         if( p == 0 )
            break;
         int l = strlen(insertStr);
         memmove(p + l, p + 1, strlen(p+1));
         memcpy(p, insertStr, l);
         p += l;
         n += l;
      }
   }

   if( isSpider ) {
      if( gratuitous.IsEmpty() ) {
      #if defined(_CONSOLE)
         cout << "gratuitous empty. (?)\n";
      #endif
      }
      else {
         buf[n] = 0;
         char *p = strstr(buf, "</BODY");
         if( p ) {
            for( int i = 0; i < 20; i++ ) {
               strcpy(p, gratuitous);
               p += gratuitous.GetLength();
            }
            strcpy(p, "</BODY></HTML>");
            n = (p - buf) + 14;
         }
         else {
         #if defined(_CONSOLE)
                  cout << "/body?\n";
         #endif
         }
      }
   }

   char temp[100];
   itoa(n, temp, 10);   // content length
   hdr += temp;
   hdr += "\r\n\r\n";

   c->write( (const char *) hdr, hdr.GetLength() );
   if( v == GET || v == POST )
      c->write(buf, n);

   return TRUE;
}
```

```
// match.cpp
//
// Ad Matching
//

#include "stdafx.h"
#include "objects.h"
#include "d:/toolkit/db.h"
#include "d:/toolkit/dbutil.h"

extern Ad *defaultAd;
extern Ad *badKeyErrorAd;

extern int nextAd;

int nAds();

// Returns TRUE if this location is in region.
//
BOOL Location::in(const Region& region)
{
   if( region.country != 0 && country != region.country )
      return FALSE;

   if( region.areaCode != 0 && areaCode != region.areaCode )
      return FALSE;

   if( !region.state.IsEmpty() && stricmp(state, region.state) != 0 )
      return FALSE;

   if( region.zipCode.IsEmpty() )
      return TRUE;

   // zip
   CString myZip = zipCode.Left(5); // strip zip+4 for now
   CString regZip = region.zipCode.Left(5);
   CString regZipEnd = region.zipEnd.Left(5);

   if( regZipEnd.IsEmpty() )
      return regZip == myZip;

   return myZip >= regZip && myZip <= regZipEnd;
}

BOOL Ad::exposuresOK(Database& db, User *user)
{
   serieeNext = 0;

   if( frequency == 0 || db == 0 )
      return TRUE;

   int n;
   BOOL found;
   {
      if( user->getID() == 0 ) {
         TRACE("userId=0?\n");
         return FALSE;
      }

      Cursor c(db);
      c.bind( SQL_C_LONG, &n, sizeof(n) );
      char sql[512] = "select exposures from exposures where ad_id=";
      addValue(sql, id, FALSE);
      strcat(sql, " and user_id=");
      addValue(sql, user->getID(), FALSE);
      c.exec(sql);
      found = c.fetchNext();
   }
   if( found ) {
      if( n >= frequency )
         return FALSE;

      serieeNext = n + 1;

      char sql[1024] =
```



```
         "update exposures set exposures=exposures+1 where ad_id=";
      addValue(sql, id, FALSE);
      strcat(sql, " and user_id=");
      addValue(sql, user->getID(), FALSE);
      db.exec(sql);

      return TRUE;
   }

   char sql[1024] =
      "insert exposures values(";
   addValue(sql, id);
   addValue(sql, user->getID(), FALSE);
   strcat(sql, ",1)");
   db.exec(sql);

   return TRUE;
}

// Note: any matching required for nontargeted ads can be placed here,
//       since this function is called for both targeting and untargeted
//       ads.
//
BOOL Ad::spreadOK(SitePage *sitepage)
{
   // is start time met?
   if( isStarted ) {
      time_t now;
      if( time(&now) < startTime )
         return FALSE;
      started = TRUE;
   }

   // impressions OK?
   if( nShown >= maxImpressions && maxImpressions != 0 )
      return FALSE;

   if( isSpreadEvenly() && ai >= 1120 )
      return FALSE;

   if( !targetSites.IsEmpty() ) {
      if( sitepage == 0 )
         return FALSE;
      BOOL v;
      BOOL found = targetSites.Lookup(sitepage->siteID, v);
      if( includeSites ) {
         // if we have pages to target too, ok if site
         // doesn't match (check if page does next).
         if( !found && targetPages.IsEmpty() )
            return FALSE;
      } else if( found )
         return FALSE;
   }

   return TRUE;
}

// Does user and site match this ad's criteria?
BOOL Ad::matches(User *user, SitePage *sitepage)
{
   if( !targetPages.IsEmpty() ) {
      if( sitepage == 0 )
         return FALSE;
      BOOL v;
      BOOL found = targetPages.Lookup(sitepage->id, v);
      if( includePages ) {
         if( !found )
            return FALSE;
      }
      else if( found )
         return FALSE;     // excluding this page
   }

   // Operating system
   DWORD o = 1 << ((int) user->os);
```

```
   if( (o & os) == 0 )
      return FALSE;

   // Browser
   o = 1 << ((int) user->browser);
   if( (o & browser) == 0 )
      return FALSE;

   // domainType
   int userISP = 0;
   int dt = (int) user->domainType;
   if( dt >= (int) dtISPOther ) {
      userISP = dt - (int) dtISPOther + 1;
      dt = 0;

      // ISP
      o = 1 << userISP;
      if( (o & isp) == 0 )
         return FALSE;
   }
   else {
      o = 1 << dt;
      if( (o & domainType) == 0 )
         return FALSE;
   }

   // location
   if( locations != 0 ) {
      if( userISP )        // if ISP, don't know location (yet)
         return FALSE;

      BOOL ok = FALSE;
      for( int i = 0; i < nLocations; i++ ) {
         if( user->location.in( locations[i] ) ) {
            ok = TRUE;
            break;
         }
      }
      if( !ok )
         return FALSE;
   }

   // hour of day / day of week
   if( hoursOfDay != 0xffffff || daysOfWeek != 0x7f ) {
      tm *t;
      if( isAbsoluteTime() ) {
         // EST time relative
         time_t now;
         time(&now);
         t = localtime(&now);
      }
      else {
         t = user->location.userRelativeTime();
         if( t == 0 )
            return FALSE;
      }
      if( (hoursOfDay & (1 << t->tm_hour)) == 0 )
         return FALSE;
      if( (daysOfWeek & (1 << t->tm_wday)) == 0 )
         return FALSE;
   }

   // sales
   if( salesVolume != 0x7fffffff ) {
      o = 1 << user->salesVolume;
      if( (o & salesVolume) == 0 )
         return FALSE;
   }

   // # employees
   if( nEmployees != 0x7fffffff ) {
      o = 1 << user->nEmployees;
      if( (o & nEmployees) == 0 )
         return FALSE;
   }
```



```
   // SIC
   if( nSICCodes ) {
      BOOL ok = FALSE;
      int i = 0;
      while( 1 ) {
         if( i >= nSICCodes ) {
            // no match
            return FALSE;
         }
         SICCode& pattern = sicCodes[i];
         user->sicCodes.reset();
         SICCode sc;
         while( user->sicCodes.getNext(sc) ) {
            if( pattern.matches(sc) ) {
               ok = TRUE;
               break;
            }
         }
         if( ok )
            break;
         i++;
      }
   }

   // Site and page categories
   // Do last, because this is expensive (disk hit)
   if( !siteCategories.isEmpty() ) {
      BOOL v;
      if( sitepage == 0 )
         return FALSE;
      sitepage->loadCategories();
      for( int i = 0; i < sitepage->categories.GetSize(); i++ )
         if( siteCategories.Lookup(sitepage->categories.GetAt(i), v) )
            return TRUE;
      return FALSE;
   }

   return TRUE;
}

inline BOOL Ad::criteriaOK(Database& db, User *user, SitePage *page)
{
   return spreadOK(page) &&
      (!isTargeted() ||
       (matches(user, page) && exposuresOK(db, user))
      );
}

// todo: if reload ads, need to handle the fact that
//       one may still be in use and can't just delete.
//       (Crit sect released during sending of file.)
//
Ad* Ad::getAd(Database& db, User *user, SitePage *page, BOOL increment)
{
   const SIMAX = 1000000;

   if( user->uniqueness < uLikely )
      return defaultAd;

   if( page == 0 ) {
      if( badKeyErrorAd )
         return badKeyErrorAd;
      ASSERT(FALSE);
   }

   if( increment )
      nextAd = (nextAd + 1) % nAds();

   int lowestSi;
   Ad *adLowestSi;

   const int start = nextAd;

   // Do a test ad, if appropriate.  Always do these first so that
```

```
#endif
    errlog.flush();
#endif
}

// temp: just return first ad (155)
//return new Ad( ado.ElementAt(0) );
return new Ad( *defaultAd );
// return 0;
}
#endif
```

```
// cookie.cpp
//

#include "stdafx.h"
#include "objects.h"

//--------------------------------------------------------------------
// Cookie

const Cookie& Cookie::operator=(const char *s)
{
   sscanf(s, "%lx", &value);
   return *this;
}

/*static*/
Cookie Cookie::alloc(DWORD userID)
{
   ASSERT( userID != 0 );
   Cookie k;
   k.value = userID;
   return k;
}

// Get value for a particular cookie name from the HTTP header
//   hdr - points to the Cookie: field in the header
//
void Cookie::getFromHeader(const char *hdr, const char *name)
{
   hdr += 7;   // skip "Cookie:"

   const char *p = strchr(hdr, '\r');
   if( p ) {
      CString nm = name;
      nm += '=';
      const char *q = strstr(hdr, nm);
      if( q && q < p )
         *this = q + nm.GetLength();
   }
}
```

```
    text << "<b>location:    </b>";
    if( location.country == 256 ) {
       text << "USA";
    } else {
       text << "country #" << location.country;
    }
    text << "\r\n";

    text << "<b>job function:</b>" << "\r\n";
    text << "<b>gender:      </b>";
    if( gender == 'm' || gender == 'M' )
       text << "Male";
    else if( gender == 'f' || gender == 'F' )
       text << "Female";
    else
       text << '?';
    text << "\r\n";
    text << "<hr>";

    Domain *d = Domain::lookupDomain(ip);
    if( d == 0 ) {
       text << "<i>No company information available.</i>";
    }
    else {
       text << "<b>domain name: </b>" << (const char *) d->domain << "\r\n";      //domainName
       text << "<b>bus. name:   </b>" << (const char *) d->name <<"\r\n";
       text << "<b>address:     </b>" << (const char *) d->address[0] << "\r\n";
       for( int i = 1; i < NADDR; i++ ) {
          if( !d->address[i].isEmpty() ) {
             text << "                " << (const char *) d->address[i] << "\r\n";
          }
       }

       text << "<b>contact:     </b>" << (const char *) d->contact[0] << "\r\n";
       for( i = 1; i < NCONTACT; i++ ) {
          if( !d->contact[i].isEmpty() ) {
             text << "                " << (const char *) d->contact[i] << "\r\n";
          }
       }

       text << "<b>industries:  </b>\r\n";
       {
          sicCodes.reset();
          SICCode sc;
          while( sicCodes.getNext(sc) ) {
             text << "                " << sc.asTextFullyPadded() << "\r\n";
          }
       }

//     for( i = 0; i < MAXSICS; i++ )
//        if( d->sicCodes[i] )
//           text << d->sicCodes[i] << ' ';

       text << "\r\n<b>no. empl.:  </b>";
       if( nEmployees )
          text << nEmployees;
       else
          text << "less than 25 (unknown)";
       text << "\r\n";
       text << "<b>revenue:     </b>";
       if( salesVolume )
          text << salesVolume;
       else
          text << "less than $1MM (unknown)";
       delete d;
    }

    text << "<hr>";
    text << "<b>You are interested in the following:</b>\r\n\r\n";
    text << "Interest Level  Category  Description\r\n";
    text << "--------------  --------  -----------------------------------------\r\n";
    {
       DWORD level;
       CString category;
```



```
       CString desc;
       Cursor c(db);
       c.bind(SQL_C_LONG, &level, 4);
       c.bind(category);
       c.bind(desc);
       char sql[512];
       wsprintf(sql,
"select interest_level,category.name from interests,user_interests\
 where interests.id=interest_id and user_id=%ld\
 order by interest_level DESC", userID);
       c.exec(sql);
       while( c.fetchNext() ) {
          char buf[32];
          wsprintf(buf, "%14ld  ", level);
          text << buf;
          text << category << "  " << desc << "\r\n";
       }
       db.commit();
    }
}

void User::_getNetworkInfo(Database& db, BOOL *timedOut)
{
    if( ip == 0 ) {
       ASSERT(FALSE);
       return;
    }

// if( domainType != dtUnknown ) {
//    // got dt from header info
//    // if ISP/OSP, location and sales, etc. don't apply.
//    // if we have done a tracert, location does apply.
//    // for ISPs/OSPs,
//    if( domainType != dtNetcom )  // did tracert for netcom
//       return;
// }

    // Note: do the following for all domain types to at least get country.

    NetworkNumber n;
    n = justNetworkNumber(ip);

    char buf[256] =
       "select domain_type,sales,num_employees,sic,country,state,zipcode,areacode from networks";
    Cursor c(db);
    if( domainType < dtAOL ) {
       c.bind(SQL_C_LONG, &domainType, sizeof(domainType));
       c.bind(SQL_C_LONG, &salesVolume, sizeof(salesVolume)); salesVolume = 0;
       c.bind(SQL_C_LONG, &nEmployees, sizeof(nEmployees)); nEmployees = 0;
       sicCodes.bind(c);
    } else {
       strcpy(buf, "select country,state,zipcode,areacode from networks where netnumber=");
    }
    strcat( buf, n.sqlStr() );

    c.bind(SQL_C_LONG, &location.country, sizeof(location.country));
    c.bind(location.state);
    c.bind(location.zipCode);
    c.bind(SQL_C_LONG, &location.areaCode, sizeof(location.areaCode));
    if( timedOut != 0 )
       c.setTimeOut(1);
    c.exec(buf);
    if( c.timedOut() )
       *timedOut = TRUE;
    else
       c.fetchNext();

    if( uniqueness == uUnknown && (int) domainType >= (int) dtAOL )
       uniqueness = uUnlikely;

    if( domainType >= dtAOL ) {
       salesVolume = 0;
       nEmployees = 0;
       sicCodes.makeNull();
       if( domainType != dtNetcom && domainType != dtISPOther ) {
```

HIGHLY CONFIDENTIAL

DC 069498

```
        // don't know location, except country
        location.state.Empty();
        location.zipCode.Empty();
        location.areaCode = 0;
      }
   }
   else {
      elcCodes.checkNull();
   }
}

#if !defined(_DERIVE)

const char cCookie[] = "Cookie:";

void User::liftVer(const char *verStr)
{
   int v1 = 0, v2 = 0;
   sscanf(verStr,
         "%d.%d", &v1, &v2);
   bVer1 = v1;
   bVer2 = v2;
}

/*
User* User::_lookupUserByID(DWORD userID)
{
   User *u = new User;
   return u;
}

User* User::_lookupUserByAddress(DWORD ip)
{
   DWORD userID = networkNodeTable.getUserID(ip, FALSE);
   if( userID == 0 ) {
      // Try to get domain info at least.  Note: if user is uniquely
      // identifiable, derive data process will create a record for the
      // user as soon as it gets a chance.
      userID = networkNodeTable.getUserID(justNetworkNumber(ip), TRUE);
   }
   if( userID ) {
      return _lookupUserByID(userID);
   }
   return 0;
}
*/

extern defaultAdsMode;

User* User::lookupUser(Database& db, DWORD ip, const char *requestHdr, BOOL loadDemographics, *
{
   BOOL _timedOut = &db == 0;
   BOOL *tmout = realTime ? &_timedOut : 0;

   //-------------------------------------------------
   // get cookie for lookup

   Cookie cookie;

   const char *ck = strstr(requestHdr, cCookie);
   if( ck )
      cookie.getFromHeader(ck, "IAF");

   //-------------------------------------------------
   // lookup

   User *u = 0;

   if( !cookie.isNull() ) {
      if( !_timedOut ) {
         u = new User;
         u->uniqueness = uYes;
         u->ip = ip;
         u->userID = cookie.value;
         u->timedOut = TRUE;
```





```
      }
      else {
         // lookup by cookie
         u = _lookupUserByID(db, cookie.value, tmout);
         if( u ) {
            u->uniqueness = uYes;
            u->ip = ip;
         }
         else {
            if( defaultAdsMode ) {
               // db conn down
               u = new User;
               u->uniqueness = uYes;
               u->ip = ip;
               u->userID = cookie.value;
            } else {
               // Couldn't find user record, we will need to
               // assign a new cookie.  Do not load by IP, because
               // we don't want this user sharing a record
               // with others without cookies.
               //
               // Note: generally, this shouldn't happen.
               cookie.value = 0;
            }
         }
      }
   }
   else if( !_timedOut ) {
      u = _lookupUserByAddress(db, ip, tmout);
      if( u ) {
         u->ip = ip;
         u->hasCookie = FALSE;
      }
   }

   if( u == 0 ) {
      // make a default user object
      u = new User;
      //u->uniqueness = uNo;
      u->ip = ip;
      u->timedOut = _timedOut;
   }

   u->headerDerive(requestHdr);

   if( !cookie.isNull() )
      u->hasCookie = TRUE;

   if( loadDemographics && !_timedOut )
      u->_getNetworkInfo(db, realTime ? &u->timedOut : 0);

   return u;
}

//-------------------------------------------------------------------------
// SitePage

Ad* Ad::findSentTo(User *user, const char *fromDoc)
{
   DWORD adNum = queryAdSent(user, fromDoc);

   for( int i = 0; i < nAds(); i++ ) {
      Ad& ad = *ads.GetAt(i);
      if( ad.id == adNum )
         return new Ad(ad);
   }

   if( badKeyErrorAd && adNum == badKeyErrorAd->id )
      return badKeyErrorAd;

   if( user->uniqueness == uLikely ) {
#if defined(errLog)
      errLog << "findSentTo failed uniqueness=likely\n";
      errLog << "  user: " << user->userID << '\n';
      errLog << "  from doc: " << fromDoc << '\n';
```

```
// objects.cpp

#include "stdafx.h"

//--------------------------------------------------------------------

const char *uniqueNames[] = {
   "Unknown", "No", "Unlikely", "Likely", "Yes"
};

const char *browserNames[] = {
   "Unknown",
   "Netscape",
   "NCSA Mosaic",
   "AOL Browser",
   "HotJava",
   "Microsoft",
   "OmniWeb",
   "Lynx",
   "NetCruiser",
   "IBM WebExplorer",
   "AIR Mosaic/Spry Mosaic",
   "MacWeb",
   "NetManage Chameleon",
   "Netsurfer",
   "Enhanced Mosaic",
   "eworld Browser",
   "Prodigy Browser",
   "Delphi Browser",
   "CNN Browser",
   "InterNotes",
   "Wollagong/ATM Emissary",
   "PipeMacWeb",
   "InternetMCI",
   "Quarterdeck Mosaic"
};

const char *osNames[] = {
   "Unknown",
   "Win16",
   "Win32",
   "Windows",
   "Win95",
   "WinNT",
   "OS/2",
   "Macintosh",
   "Mac 68k",
   "Mac PowerPC",
   "Unix (brand unknown)",
   "Unix (other)",
   "Unix (Sun)",
   "Unix (Linux)",
   "Unix (HP)",
   "Unix (AIX)",
   "Unix (OSF)",
   "Unix (IRIX)",
   "NEXT",
   "Unix (SGI)"
};

const char *domainTypeNames[] = {
   "Unknown",
   "Commercial", "Education", "Government",
   "Military", "K-12", "Foreign", "Networks",
   "Organizations",

   0,

   "AOL",
   "Prodigy",
   "Compuserve",
   "Delphi",
   "eworld",
   "MSN",
   "DowJones",
```



```
   "Genie",

   0,0,0,0,0,0,0,
   "Reserved for ISP Names"
};

const char *ISPNames[] = {
   "ISP",
   "NetCom",
   "PSI",
   "UUNet",
   "Advantis",
   "Concentric Research Corp.",
   "CRL",
   "MCI",
   "Portal Information Network"
};

const char *salesStr[] = {
   "unknown",
   "$1 - $49,999",
   "$50,000 - $99,999",
   "$100,000 - $249,999",
   "$250,000 - $499,999",
   "$500,000 - $999,999",
   "$1 million - $4,999,999",
   "$5 million - $9,999,999",
   "$10 million - $49,999,999",
   "$50 million - $499,999,999",
   "$500 million - $999,999,999",
   "$1 billion and over"
};

const char *empStr[] = {
   "unknown",
   "1 - 4",
   "5 - 9",
   "10 - 14",
   "15 - 19",
   "20 - 49",
   "50 - 99",
   "100 - 499",
   "500 - 999",
   "1,000 and over",
};

const char *genderStr[] = {
   "unknown",
   "Male",
   "Female"
};

const char *timesStr[] = {
   "12am-1am",
   "1am-2am",
   "2am-3am",
   "3am-4am",
   "4am-5am",
   "5am-6am",
   "6am-7am",
   "7am-8am",
   "8am-9am",
   "9am-10am",
   "10am-11am",
   "11am-12pm",
   "12pm-1pm",
   "1pm-2pm",
   "2pm-3pm",
   "3pm-4pm",
   "4pm-5pm",
   "5pm-6pm",
   "6pm-7pm",
   "7pm-8pm",
   "8pm-9pm",
   "9pm-10pm",
```

```
    "10pm-11pm",
    "11pm-12am"
};

const char *dowStr[] = {
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
};


#if !defined(_JUSTSTRINGS)


#include <strstrea.h>
#include <fstream.h>
#include "winsock.h"
#include "objects.h"
#include "tables.h"
#include "/d/toolkit/isf_util.h"
#include "/d/toolkit/db.h"
#include "/d/toolkit/dbutil.h"
#include "/d/derive/eqlderive.h"
#include "/d/newderive/elg.h"
#include "rememberad.h"

extern ofstream errLog;

extern Ad *badKeyErrorAd;

int nextAd = 0;

#if !defined(_DERIVE)
int nAds();
#endif
extern Ad *defaultAd;

//---------------------------------------------------------------------
// User

BOOL User::cookieCapable() const
{
    // todo: add new version of Internet Explorer

    return browser == brNetscape &&
        (bVer2 >= 1 ||
        bVer1 > 1);
}

void User::deriveDomainTypeFromBrowser()
{
    switch( browser ) {
        case brAOL:
            domainType = dtAOL;
            break;
        case brEWorld:
            domainType = dtEWorld;
            break;
        case brProdigy:
            domainType = dtProdigy;
            break;
        case brDelphi:
            domainType = dtDelphi;
            break;
        default:
            ;
    };
}

DWORD User::getID() const
{
```





```
    return userID;
}

User::User()
{
    timedOut = FALSE;
    userID = 0;
    uniqueness = uUnknown;
    ip = 0;
    browser = brUnknown;
    bVer1 = bVer2 = 0;
    os = osUnknown;
    domainType = dtUnknown;
// for( int i = 0; i < MAXSICS; i++ )
//     sicCodes[i] = 0;
    nEmployees = 0;
    salesVolume = 0;
    proxy = FALSE;
    isNetworkDescription = FALSE;
    ftpTried = FALSE;
    hasCookie = FALSE;
}

void User::describe(Database& db, strstream& text)
{
    in_addr ipAddr = (in_addr&) ip;
    text << "<b>ip:          </b>"
         << (int) ipAddr.S_un.S_un_b.s_b1 << '.'
         << (int) ipAddr.S_un.S_un_b.s_b2 << '.'
         << (int) ipAddr.S_un.S_un_b.s_b3 << '.'
         << (int) ipAddr.S_un.S_un_b.s_b4 << "\r\n";

    gender = ' ';
    if( !emailAddr.IsEmpty() ) {
        // get name/gender
        Cursor c(db);
        CString g,f,l;
        c.bind(g);
        c.bind(f);
        c.bind(l);
        Signature s;
        s.email = emailAddr;
        s.l.nullEmail();
        if( !s.l.domain.IsEmpty() ) {
            char sql[1024] = "select gender,fname,lname from listings where emailname=";
            addValue(sql, s.l.emailname, FALSE);
            strcat(sql, " and domain=");
            addValue(sql, s.l.domain, FALSE);

            c.exec(sql);
            if( c.fetchNext() ) {
                if( g.GetLength() )
                    gender = g.GetAt(0);
                name = f + ' ' + l;
                cap(name);
            }
        }
        db.commit();
    }

    text << "<b>unique:      </b>" << uniqueNames[uniqueness] << "\r\n";
    text << "<b>cookie:      </b>" << (hasCookie ? "Yes" : "No") << "\r\n";
    text << "<b>browser:     </b>" << browserNames[browser] << "\r\n";
    text << "<b>browser ver: </b>" << (int) bVer1 << '.' << (int) bVer2 << "\r\n";
    text << "<b>os:          </b>" << osNames[os] << "\r\n";
    text << "<b>domain type: </b>" << domainTypeNames[domainType] << "\r\n";
    text << "<b>proxy:       </b>" << (proxy ? "Yes" : "No") << "\r\n";
    text << "<hr>";
    text << "<b>name:        </b>" << name << "\r\n";
    text << "<b>title:       </b>" << title << "\r\n";
    text << "<b>state:       </b>" << location.state << "\r\n";
    text << "<b>zip code:    </b>" << location.zipCode << "\r\n";
    text << "<b>area code:   </b>" << location.areaCode << "\r\n";
    text << "<b>phone:       </b>" << phone << "\r\n";
    text << "<b>e-mail:      </b>" << emailAddr << "\r\n";
```

```
Ad *ad = Ad::findSentTo(user, from);

if( ad == 0 ) {      // fix!
   delete user;
   return;
}

SitePage *page = SitePage::lookupPage(from, request);

CString hdr =
   "HTTP/1.0 200 OK\r\nContent-Type: text/html\r\nPragma: no-cache\r\nContent-Length: ";
char buf[32000];
*buf = 0;
strstream text(buf, 32000, ios::out);

// fill content
text << "<html><body><h1>Jump Redirect</h1>";
text << "<pre>";
text << "jumping from document: " << (from << "\r\n";
text << "       would jump to: "
    << "<a href=\"" << (const char *) ad->jumpTo << "\">"
    << (const char *) ad->jumpTo << "</a>\r\n\r\n";
text << "                  ad: " << (const char *) ad->fullName() << "\r\n\r\n";

CString fn = ad->fileName;
text << "<center><img src=\"/"
    << (const char *) fn
    << "\">";
text << "</pre></body></html>";

int n = text.pcount();
char temp[100];
itoa(n, temp, 10);   // content length
hdr += temp;
hdr += "\r\n\r\n";

c->write( (const char *) hdr, hdr.GetLength() );
c->write(buf, n);

logJump(ad, user, page);

delete page;
delete ad;
delete user;
*/
}

void GetRequest::sendFrame(const char *from)
{
   CString s = "HTTP/1.0 200 OK\r\nContent-Type: text/html\r\n";
   s += "\r\n<HTML><BODY><CENTER><a href=\"http://206.4.219.5/jump/";
   s += from;
   s += "\">\n<img src=\"http://206.4.219.5/ad/";
   s += from;
   s += "\">\n</a></center></html>";   //  Width=468 Height=60
   c->write( (const char *) s, s.GetLength() );
}

void GetRequest::activity(const char *activityStr)
{
   // go ahead and send for best response time
   sendFile("c:\\lan\\html\\dot.gif");

   BOOL bad = FALSE;

   // send the file first
   ActivityType type;
   CString siteKey;
   BOOL ok = TRUE;
   switch( *activityStr )
   {
      case 'a':
         type = Interest;
         break;
      case 'i':
         type = InfoRequest;
         break;
      case 's':
         type = Sale;
         break;
      default:
         ok = FALSE;
   }

   if( ok ) {
      const char *p = activityStr + 1;
      if( *p != '/' )
         ok = FALSE;
      else {
         p++;
         const char *q = strchrip, '/');
         if( q == 0 )
            ok = FALSE;
         else
            siteKey = CString(p, q - p);
      }
   }

   if( ok ) {
      Database *db = getFromPool();
      User *user = User::lookupUser(*db, userIP, request);
      DWORD advertiserID = 0;
      // todo: fix if not assigned a user ID. (use IP)?
      if( user->userID != 0 ) // if not from LAN, skip logging
      {
         Cursor c(*db);
         c.bind(SQL_C_LONG, &advertiserID, sizeof(advertiserID));
         char sql[1024] = "select id from advertisers where sitekey=";
         addValue(sql, siteKey, FALSE);
         c.exec(sql);
         ok = c.fetchNext();
      }

      db->commit();

      if( ok ) {
         ++activity;
         if( advertiserID != 0 )
            logActivity(user, advertiserID, type);
      }

      delete user;
      releaseToPool(db);
   }

   if( !ok ) {
      message( CString("Invalidate activity str: ") +
               CString(activityStr).Left(80) );
//    sendError(c, "404 Not Found");
   }
}

void GetRequest::sendAd(const char *from)
{
   if( from && strnicmp(from, "www.", 4) == 0 )
      from += 4;

   Database *db = getFromPoolTimeout();

   static DWORD lastFTP;
   startLatency = GetTickCount();

   User *user;
   SitePage *page;
   Ad *ad;

   user = User::lookupUser(*db, userIP, request, TRUE, TRUE);
   if( db == 0 ) {
      page = 0;
   }
```

```
   else {
      page = SitePage::lookupPage(*db, from, request);
   }
   ad = Ad::getAd(*db, user, page, v == GET);

// if( v == GET ) {
//    TRACE("get %s\n", from);
// }

   static int randCutoff = 0; //RAND_MAX / 4;

   BOOL doFTP = !user->tempUserObject() &&
      !user->ftpTried && user->uniqueness >= uLikely && !user->proxy &&
      rand() < randCutoff && (startLatency - lastFTP > 6000);
   DWORD dw;
   if( doFTP )
      dw = WaitForSingleObject(ftpMutex, 0);
   if( doFTP && dw != WAIT_FAILED && dw != WAIT_TIMEOUT ) {

      lastFTP = startLatency;

      // Remember that we're doing FTP for user.  Only do once.
      user->ftpTried = TRUE;
      user->updateFTPTried(*db);

      // Redirect
      CString s = "Location: ";
      s += "ftp://206.4.219.6/";
      char buf[10];
      wsprintf(buf, "%lx", user->getID());
      s += buf;
      s += '/';
      CString fn = ad->getFileName();
      s += (const char *) fn;

      errLog << "Trying FTP\n";
      errLog << "  user " << user->getID() << '\n';
      errLog << "  browser " << browserNames[(int) user->browser] << '\n';
      errLog << "  url " << s << '\n';

      s += "\r\n";
      sendError(c, "302 Moved Temporarily", s);

      VERIFY( ReleaseMutex(ftpMutex) );

      logAdSend(ad, user, page);

      errLog.flush();

      db->commit();
      releaseToPool(db);
   }
   else
   {
      //{.cs.leave();
      send(*db, ad, user);    // this function calls releaseToPool()
      //{.cs.enter();
      if( v == GET ) {
         static int counter;
         if( ++counter & 3 )
            ad->calcSI();     // update SI every 4 or so deliveries

         rememberSend(ad, user, from);
         logAdSend(ad, user, page);
         if( user->timedOut ) {
            if( db == 0 )
               poolTimeOuts++;
            else
               timeOuts++;
         }

         // state
         c->close();    // flush send
         DWORD endSend = GetTickCount();
         la      (endLatency - startLatency);
```



```
         adSendTimeMax(endSend - startLatency);
      }
   }

// delete ad;
   delete page;
   delete user;
}

void GetRequest::takeJump(const char *_from)
{
   Database &db = *getFromPool();

// jumpingWhere(from);
// return;

   User *user = User::lookupUser(db, userIP, request, FALSE);

   if( _from && strnicmp(_from, "www.", 4) == 0 )
      _from += 4;

   CString from;
   {
      const char *p = strchr(_from, '?');
      if( p == 0 ) {
         from = _from;                                    \
         char buf[512];
         wsprintf(buf, "no lsmap %d: %s", user == 0 ? 999 : (int) user->browser, (const char *
         message(buf);
      }
      else
         from = CString(_from, p - _from);
   }

   Ad *ad = Ad::findSentTo(user, from);
   SitePage *page = SitePage::lookupPage(db, from, request);

   ///{.cs.leave();
   CString s = "Location: ";
   s += ad->jumpTo(// + "?from=laf";
   s += "\r\n";
   sendError(c, "301 Moved Permanently", s);
   c->close();
   //{.cs.enter();

   // Must do this so activity will be logged properly.
   // See GetRequest::activity().
   user->makePermanent(db);

   logJump(ad, user, page);

   delete page;
   delete ad;
   delete user;
   db.commit();
   releaseToPool(&db);
}
```

```
// getrequest.cpp
//

#include "stdafx.h"
#include <strstrea.h>
#include <fstream.h>
#include "d/toolkit/sock.h"
#include "getrequest.h"
#include "rememberad.h"
#include "d/toolkit/iaf_util.h"
#include "log.h"
#include "status.h"
#include "d/toolkit/crit.h"
#include "d/toolkit/db.h"
#include "d/toolkit/dbutil.h"
#include "d/toolkit/dbpool.h"

extern CriticalSection fast;
//extern Database iafmain;

extern ofstream errLog;
extern int activity;

extern const char *browserNames[];

const char *progName = "AdSvr";

void message(const char *);

void recalcSI();

DWORD startLatency, endLatency;

// This used to prevent multiple concurrent FTP
// requests right now because our FTPD implementation
// only does one at a time.
//
extern HANDLE ftpMutex;

void GetRequest::service()
{
   const char *p = strchr(request, ' ');
   if( p )
      fileName = CString(request, p - request);
   else
      fileName = request;

   if( fileName.Left(4) == "/ad/" )
      sendAd((const char *) fileName + 4);
   else if( fileName.Left(9) == "/adframe/" )
      sendFrame((const char *) fileName + 9);
   else if( fileName.Left(6) == "/jump/" )
      takeJump((const char *) fileName + 6);
   else if( fileName.Left(10) == "/activity/" )
      activity((const char *) fileName + 10);
   else if( fileName.Left(7) == "/whoami" ) {
      //Crit c(fast);
      whoAmI();
   }
   else if( fileName.Left(8) == "/viewad/" ) {
      CString asFileName;
      asFileName.Format( "c:/ian/ads/%s", (LPCTSTR)fileName+8);
      sendFile( asFileName );
   }
   else if( fileName.Left(11) == "/state.htm?" ) {
      sendError(c, "404 Not Found: Results forecast moved to another server");
      //state((const char *) fileName + 11);
   }
   else if( fileName.Left(10) == "/sendinfo/" ) {
      sendInfo((const char *) fileName + 10);
      return;
   }
   else if( fileName.Left(4) == "/si_" ) {
      // send info stuff
      si_((const char *) fileName + 4);
```





```
   }
   else if( fileName.Left(9) == "/sysstate" ) {
      sysState();
   }
   else {
      const char *p = fileName;
      if( strnicmp(p, "/java/", 6) == 0 ) {
         if( strstrip, "..") == 0 )
            sendFile(p);
         else
            sendError(c, "404 Not Found");
      }
      else {
         if( *p == '/' )
            p++;
         if( *p == 0 ) {
            // send default
            sendFile("c:\\ian\\html\\default.htm");
            return;
         } else {
            if( //strchrip, '/') == 0 && strchrip, '\\') == 0 &&
               strstrip, "..") == 0 )
            {
               CString f = "c:\\ian\\html\\";
               f += p;
               sendFile(f);
               return;
            }
         }
         sendError(c, "404 Not Found");
      }
   }

   // Normally we adjust SI for an ad as it is delivered.
   // However, occasionally should do all ads in case one hasn't
   // been delivered but time has passed.
   static int counter;
   if( ++counter > 200 ) {    // adjust constant as traffic increases
      counter = 0;
      Crit c(fast);
      if( allFree() ) {
         recalcSI();    // recalc SI for all ads
      }
      else {
         counter = 175;    // try again soon
      }
   }
}

const char cHeader[] =
   "HTTP/1.0 200 OK\r\nContent-Type: image/gif\r\nContent-Length: ";

// send() should commit the DB if it does any DB operations because
// the caller commits ahead of time so that the transaction won't
// remain open while the file is sent.
//
void GetRequest::send(Database& db, Ad *ad, User *u)
{
   CString hdr = cHeader;

   const BUFSIZE = 32000;
   char buf[BUFSIZE];

   Cookie sendCookie;
   if( idb != 0 ) {
      if( !u->hasCookie ) {
         if( u->cookieCapable() && !u->timedOut )
         {
            // If a user record already exists, it's probably because
            // this IP address is shared with other users (proxy, IP pool,
            // etc.)  So, we want to create another record; we don't want
            // to assign the same cookie to different people!
            u->userID = 0; // create new record

            // generate a cookie for the user
```

```
            u->hasCookie = TRUE;
            u->makePermanent(db);
            sendCookie.value = u->getID();
         }
      }

      // release DB here so that we don't keep a db connection occupied
      // while sending the ad
      db.commit();
      releaseToPool(db);
   }

   CFile f;
   int n = 0;
   if( v == GET ) {
      CString s = ad->fullName();
      if( !f.Open(s, CFile::modeRead | CFile::shareDenyWrite) ) {
         message( CString("couldn't open ") + s );
         TRACE("couldn't open %s\n", (const char *) s);
         ASSERT(FALSE);
         return;
      }

      n = f.Read(buf, BUFSIZE);
      ASSERT( n != 0 && n != BUFSIZE );
   }
   else {
      n = getFileSize( ad->fullName() );
      // next line is a test for NCSA Mosaic HEAD
      //n = 1;
   }

   char temp[100];
   itoa(n, temp, 10);   // content length
   hdr += temp;
   if( !sendCookie.IsNull() ) {
      wsprintf(temp,
            "\r\nSet-Cookie: IAP=%lx; path=/; expires=Wed, 09-Nov-99 23:59:00 GMT",
            sendCookie.value);
      hdr += temp;
   }

   // last-modified time
   hdr += "\r\nLast-Modified: " + curHTTPTime();

   //test
// hdr += "\r\nPragma: no-cache";

   hdr += "\r\n\r\n";

   endLatency = GetTickCount();

   c->write( (const char *) hdr, hdr.GetLength() );
   if( v == GET )
      c->write(buf, n);
}

// diagnostic
void GetRequest::sysState()
{
   static char *typeStrs[] = {
      "Normal",
      "Test",
      "Barter",
      "Ian Dev" };


   CString hdr =
      "HTTP/1.0 200 OK\r\nContent-Type: text/html\r\nContent-Length: ";
   char buf[32000];
   *buf = 0;
   strstream text(buf, 32000, ios::out);

   // fill content
   text << "<html><body bgcolor=#ffffff>\r\n";
```





```
   text << "<h1>System State</h1><p>";
   text << "<table border=1 cellpadding=3>";
   text << "<tr><td><b>Name</b></td><td><b>Type</b></td><td><b>Sl</b></td>";
   text << "<td><b>Ads Sent</b></td><td><b>Ads Booked</b></td></tr>\n";

   // Get a db connection to lock the ads array so that
   // it isn't reloaded or anything while we are processing.
   Database *db = getFromPool();

   for( int i = 0; i < ads.GetSize(); i++ ) {
      Ad *ad = ads.GetAt(i);
      text << "<tr><td><a href=\"http://ad.iantargets.com/viewad/";
      ad->fileName.MakeLower();
      text << ad->fileName << "\">" << ad->fileName << "</td>";
      text << "<td>" << typeStrs[ad->type] << "</td>";
      text << "<td>" << ad->sl << "</td>";
      text << "<td>" << ad->nShown << "</td>";
      text << "<td>" << ad->maxImpressions << "</td></tr>\n";
   }

   releaseToPool(db);

   text << "</table>";
   text << "</body></html>";

   int n = text.pcount();
   char temp[100];
   itoa(n, temp, 10);   // content length
   hdr += temp;
   hdr += "\r\n\r\n";

   c->write( (const char *) hdr, hdr.GetLength() );
   c->write(buf, n);
}

// diagnostic
void GetRequest::whoAmI()
{
   Database *db = *getFromPool();

   User *user = User::lookupUser(db, userIP, request);
   user->lookupAncillaryInfo(db);

   CString hdr =
      "HTTP/1.0 200 OK\r\nContent-Type: text/html\r\nContent-Length: ";
   char buf[32000];
   *buf = 0;
   strstream text(buf, 32000, ios::out);

   // fill content
   text << "<html><body bgcolor=#ffffff><h1><IMG SRC=\"ianlogos.gif\" ALIGN=\"BOTTOM\">User In
   text << "<pre>";
   user->describe(db, text);
   text << "</pre></body></html>";

   int n = text.pcount();
   char temp[100];
   itoa(n, temp, 10);   // content length
   hdr += temp;
   hdr += "\r\n\r\n";

   c->write( (const char *) hdr, hdr.GetLength() );
   c->write(buf, n);

   delete user;
   releaseToPool(db);
}

// diagnostic
void GetRequest::jumpingWhere(const char *from)
{
   ASSERT(FALSE);
   // fix for multi-db conns
/*
   User *user = User::lookupUser(userIP, request, FALSE);
```

```
    else {
       check(userAgent, "OmniWeb", brOmniWeb, osNEXT);
       check(userAgent, "Lynx", brLynx, osUnknown);
       check(userAgent, "IBM WebExplorer", brWebExplorer, os0S2);
       check(userAgent, "AIR_Mosaic", brAirMosaic, osWin);
       check(userAgent, "SPRY_Mosaic", brAirMosaic, osWin);
       check(userAgent, "MacWeb", brMacWeb, osMac);
       check(userAgent, "NetManage", brChameleon, osWin);
       check(userAgent, "Netsurfer", brNetsurfer, osNEXT);
       check(userAgent, "GNNworks", brGNN, osWin);
       check(userAgent, "InterNotes", brNotes, osUnknown);
       check(userAgent, "Emissary", brEmissary, osUnknown);
       check(userAgent, "PipeMacWeb", brPipeMacWeb, osMac);
       check(userAgent, "InternetMCI", brMCI, osUnknown);
       check(userAgent, "Quarterdeck", brQuarterdeck, osUnknown);
       check(userAgent, "NCSA Mosaic (for the X ", brNCSA, osUnixUnknown);
       if( check(userAgent, "eworldbrowser", brEWorld, osMac) ) {
          if( userAgent.Find("68K") >= 0 )
             os = osMac68;
          else if( userAgent.Find("PPC") >= 0 )
             os = osMacPPC;
          uniqueness = uNo;
          domainType = dtEWorld;
       }
       else if( check(userAgent, "PRODIGY", brProdigy, osUnknown) ) {
          uniqueness = uNo;
          domainType = dtProdigy;
       }
       else if( check(userAgent, "Delphi", brDelphi, osUnknown) ) {
          uniqueness = uNo;
          domainType = dtDelphi;
       }
       else if( browser == brUnknown ) {
          TRACE("unknown useragent: %s\n", (const char *) userAgent);
          if(tOS(userAgent));
       }
    }

    if( userAgent.Find("via proxy") >= 0 ) {
       proxy = TRUE;
       if( uniqueness == uUnknown )
          uniqueness = uNo;
    }
 }
}
```

```
// location.cpp

#include "stdafx.h"
#include "objects.h"
#include "/d/toolkit/mapstate.h"
#include "/d/toolkit/tzutil.h"

// next line should be in tzutil.h
extern CountryTimeZoneMap mapCountryTimeZones;

struct IsDaylightSavings
{
   IsDaylightSavings()
   {
      TIME_ZONE_INFORMATION i;
      DWORD r = GetTimeZoneInformation(&i);
      daylightSavings = r == TIME_ZONE_ID_DAYLIGHT;
   }

   BOOL daylightSavings;
} ids;

tm* Location::userRelativeTime( time_t timeRelative )
{
   int utc_offset;
   int daylight_bias;

   if( country == 256 ) {
      if( !getStateTimeZoneInfo(state, utc_offset, daylight_bias) )
         return FALSE;
   }
   else if( country == 0 ) {
      return FALSE;
   }
   else {
      DWORD dwBias;
      if( !mapCountryTimeZones.Lookup(country, dwBias) )
         return FALSE;

      utc_offset = LOWORD(dwBias);
      daylight_bias = HIWORD(dwBias);
   }

   time_t ttTime;

   //
   // if timeRelative == 0, this assumes that they want the time
   // relative to the current time
   ttTime = timeRelative;
   if (!ttTime)
   {
      time(&ttTime);
   }

   if( ids.daylightSavings && daylight_bias != TZ_BIAS_UNDEFINED )
      ttTime += daylight_bias * 60 * 60;
   else
      ttTime += utc_offset * 60 * 60;

   return gmtime(&ttTime);
}
```

```
/ request.h
/

if !defined(_REQUEST_H_)
define _REQUEST_H_

include "d:\toolkit\sock.h"

num Verb { UNKNOWN, GET, HEAD, POST };

lass Connection;

lass Request

ublic:
   Request(Connection *c, Verb v,
           const char *requestText,
           const sockaddr_in& from);

   virtual void service();

   DWORD getIP() const { return userIP; }
   const char* getRequest() const { return request; }
   Connection* getConn() const { return c; }

   void sendInternalError();

protected:
   BOOL sendFile(const char *fileName, const char *insertStr = 0);

   Connection *c;
   const char *request;
   Verb v;
   CString fileName;
   DWORD userIP;
};

void sendError(Connection *c, const char *msg, const char *headerField = 0);

#endif
```

```
// header.cpp
//

#include "stdafx.H"
#include "objects.h"
#include "/d/toolkit/iaf_util.h"

const char cBrowser[] = "User-Agent:";

void message(const char *);

BOOL User::check(CString& userAgent, const char *pat, Browser b, OS o)
{
   if( browser != brUnknown )
      return FALSE;

   int l = strlen(pat);
   if( userAgent.Left(l) == pat ) {
      browser = b;
      os = o;
      const char *p = userAgent;
      p += l;
      p = strchr(p, '/');
      if( p ) {
         liftVer(p + 1);
      }
      return TRUE;
   }

   return FALSE;
}

static void match(OS& os, const char *userAgent, const char *pat, OS o)
{
   if( strstr(userAgent, pat) != 0 )
      os = o;
}

void User::liftOS(const CString& userAgent)
{
   if( userAgent.Find("X11") >= 0 ) {
      os = osUnixOther;
      match(os, userAgent, "SunOS", osUnixSun);
      match(os, userAgent, "HP-UX", osUnixHP);
      match(os, userAgent, "Linux", osUnixLinux);
      match(os, userAgent, "OSF", osUnixOSF1);
      match(os, userAgent, "AIX", osUnixAIX);
      match(os, userAgent, "IRIX", osUnixIRIX);
   }
   else if( userAgent.Find("Windows") >= 0 ) {
      if( userAgent.Find("32bit") >= 0 ||
          userAgent.Find("95") >= 0
        )
      {
         os = osWin32;
      } else {
         os = osWin16;
      }
   }
   else if( userAgent.Find("Win95") >= 0 ) {
      os = osWin95;
   }
   else if( userAgent.Find("Win16") >= 0 ) {
      os = osWin16;
   }
   else if( userAgent.Find("Macintosh") >= 0 ) {
      os = osMac;
      match(os, userAgent, "PPC", osMacPPC);
      match(os, userAgent, "68K", osMac68);
   }
   else if( userAgent.Find("WinNT") >= 0 ) {
      os = osWinNT;
   }
   else {
```





```
      s += userAgent.Left(70);
      message(s);
   }
}

// derive information about the user from the request header
//
void User::headerDerive(const char *requestHeader)
{
   const char *ua = strstr(requestHeader, cBrowser);
   if( ua == 0 ) {
      // if no user agent field, something weird we
      //  don't know much about, don't assume unique.
      uniqueness = uUnlikely;
   }
   else {
      ua += 11;
      while( *ua == ' ' )
         ua++;
      const char *p = strchr(ua, '\r');
      if( p ) {
         CString userAgent(ua, p - ua);
         if( userAgent.Left(8) == "Mozilla/" ) {
            browser = brNetscape;
            liftVer((const char *) userAgent + 8);

            // OS
            liftOS(userAgent);
         }
         else if( userAgent.Left(12) == "NCSA Mosaic/" ) {
            browser = brNCSA;
            liftVer((const char *) userAgent + 12);

            // OS
            match(os, userAgent, "Windows", osWin);
            match(os, userAgent, "X11", osUnixUnknown);
            match(os, userAgent, "X Window", osUnixUnknown);
         }
         else if( strncmp(userAgent, "IWENG/", 6) == 0 ) {
            browser = brAOL;
            uniqueness = uNo;
            domainType = dtAOL;
            liftVer((const char *) userAgent + 6);
            os = osWin;
         }
         else if( strncmp(userAgent, "AOLbrowser/", 10) == 0 ) {
            browser = brAOL;
            uniqueness = uNo;
            domainType = dtAOL;
            liftVer((const char *) userAgent + 11);
            os = osMac;
         }
         else if( userAgent.Left(28) == "Microsoft Internet Explorer/" ) {
            // Microsoft Internet Explorer/4.40
            browser = brMicrosoft;
            liftVer((const char *) userAgent + 28);
            os = osWin32;
            match(os, userAgent, "Windows 95", osWin95);
         }
         else if( userAgent.Left(8) == "HotJava/" ) {
            browser = brHotJava;
            liftVer((const char *) userAgent + 8);
         }
         else if( userAgent.Left(16) == "Enhanced_Mosaic/" ) {
            browser = brEnhancedMosaic;
            liftVer((const char *) userAgent + 16);
            os = osWin;
            if( userAgent.Find("Win32") >= 0 )
               os = osWin32;
         }
         else if( userAgent.Left(11) == "NetCruiser/" ) {
            liftVer((const char *) userAgent + 11);
            browser = brNetCruiser;
            os = osWin;
         }
```

```
// server.h
//
// General sd server startup stuff.
//

BOOL startServer();
```

```
// status.h

void setStatus(const char *s);

extern int adsSent;
extern int jumpsTaken;
extern int totalAdSendLatency;
extern int totalAdSendTime;
extern int timeOuts;
extern int poolTimeOuts;

extern int barter, lanDev, testAd;

void latencyWas(int n);
void adSendTimeWas(int n);

void adSent();
```

EXHIBIT B

DX 50

HIGHLY CONFIDENTIAL

DC 069484

REQUEST.H 11-Jan-1996 13:25 Page 1(1)

getrequest.h

```
( !defined(_GETREQUEST_H_)
efine _GETREQUEST_H_

nclude "request.h"
nclude "objects.h"

ass GetRequest : public Request

blic:
  GetRequest(Connection *c, Verb v,
             const char *requestText,
             const sockaddr_in &from) :
    Request(c, v, requestText, from) { }

  virtual void service();

otected:
  void whoAmI();
  void jumpingWhere(const char *from);
  void sendAd(const char *from);
  void activity(const char *activityStr);
  void sendFrame(const char *from);      // Netscape 2.0 frames
  void takeJump(const char *from);
  void eyeState();

  void sendDatabase& db, Ad *ad, User *u);

  // send info
  void sendInfo(const char *url);
  void el_(const char *url);

ndif
```

```
// rememberad.h
//

void rememberSend(Ad *ad, User *u, const char *fromDoc);

// returns Ad ID
DWORD queryAdSent(User *u, const char *fromDoc);
```